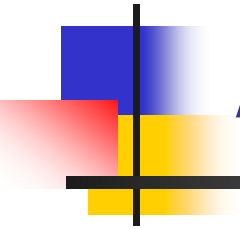
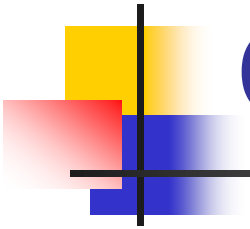


COMP 6231: Distributed System Design

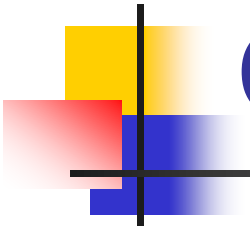
Distributed Objects and CORBA

Based on Chapter 8 of the textbook and the slides from Prof. M.L. Liu, California Polytechnic State University



CORBA

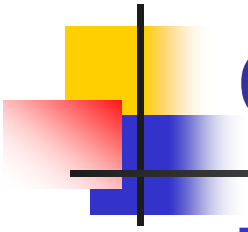
- The Common Object Request Broker Architecture (CORBA) is a standard architecture for a distributed objects system.
- CORBA is designed to allow distributed objects to interoperate in a heterogenous environment, where objects can be implemented in different programming language and/or deployed on different platforms



Goal of CORBA

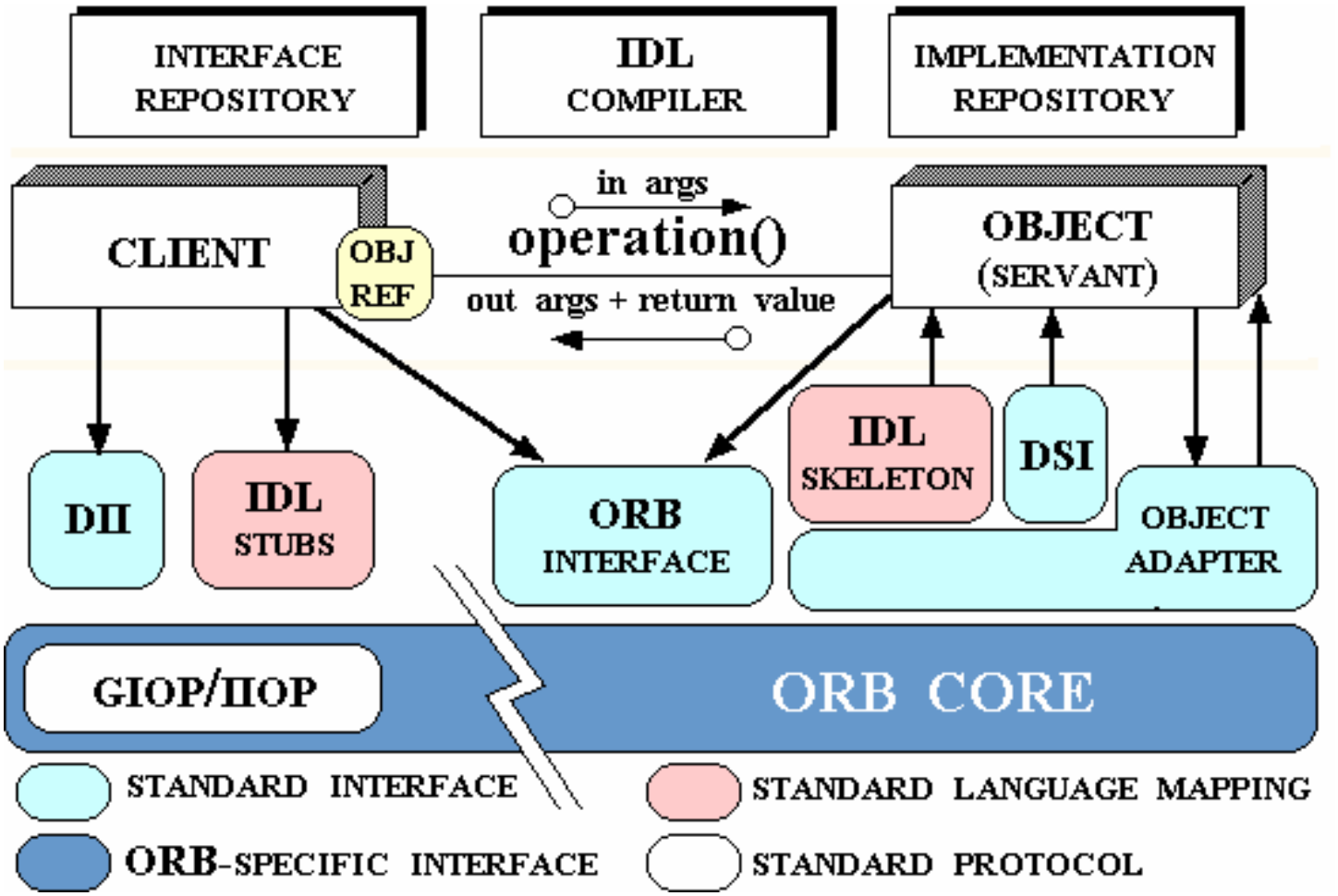
- Support distributed and heterogeneous object request in a way transparent to users and application programmers
- Facilitate the integration of new components with legacy components
- Open standard that can be used free of charge
- Based on industry-wide consensus

CORBA design goals/characteristics

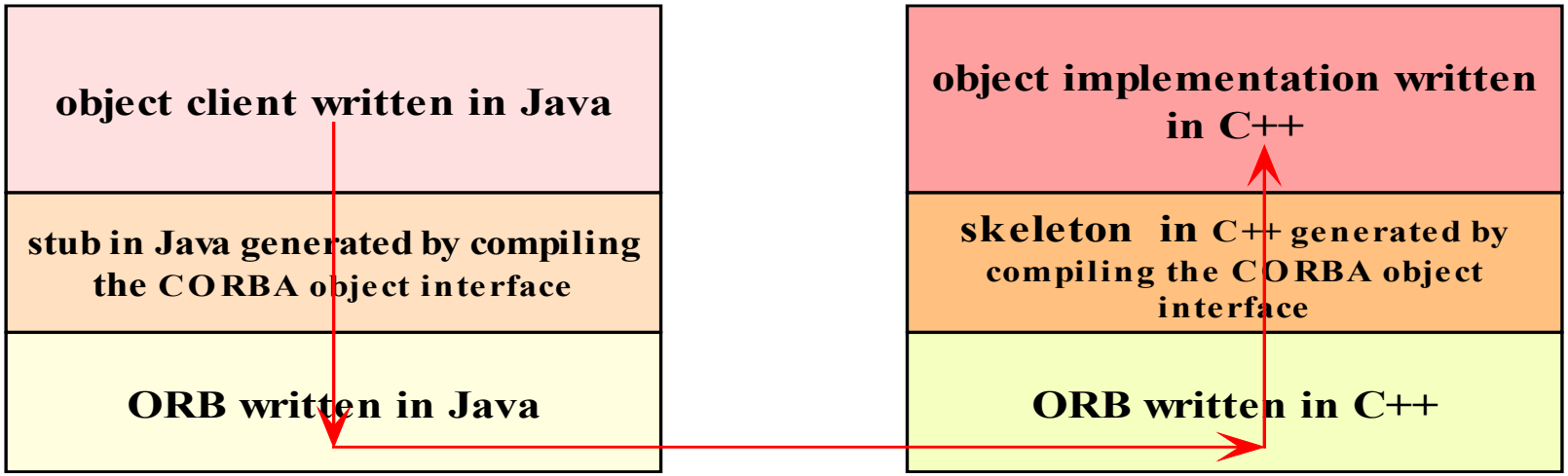


- No need to pre-determine:
  - The programming language
  - The hardware platform
  - The operating system
  - The specific object request broker
  - The degree of object distribution
- Open Architecture:
  - Language-neutral Interface Definition Language (IDL)
  - Language, platform and location transparent
- Objects could act as clients, servers or both
- The Object Request Broker (ORB) mediates the interaction between client and server objects

ORB Architecture

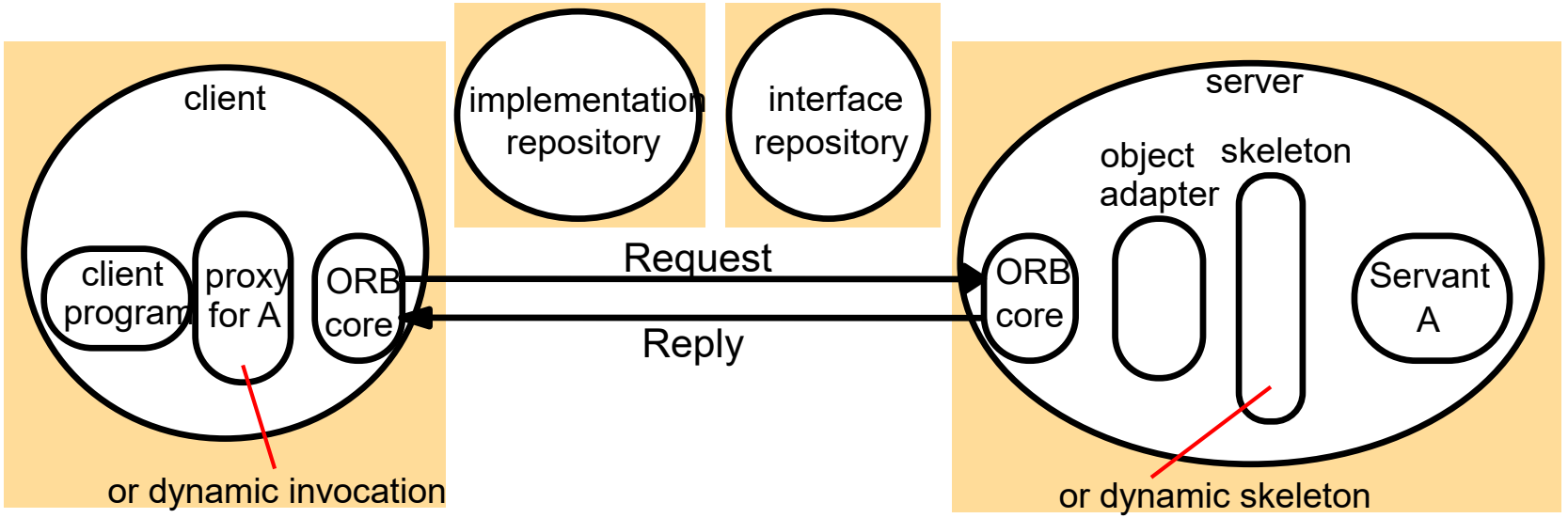


Cross-language CORBA application

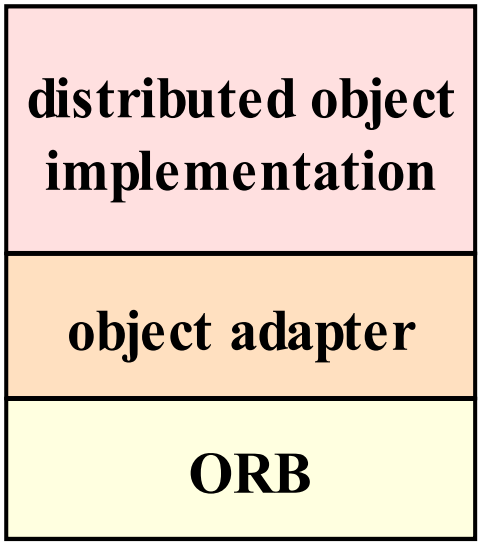


The main components of the CORBA architecture

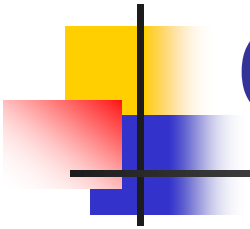
- The CORBA architecture is designed to allow clients to invoke methods in CORBA objects
  - Clients and objects can be implemented in a variety of programming languages
  - It has the following additional components compared to RMI
    - Object adapter, implementation repository and interface repository



Object Adapter

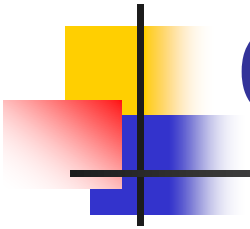


- An object adapter simplifies the responsibilities of an ORB by assisting the ORB in delivering a client request to an object implementation.
- When an ORB receives a client's request, it locates the object adapter associated with the object and forwards the request to the adapter.
- The adapter interacts with the object implementation's skeleton, which performs data marshalling and invokes the appropriate method in the object.



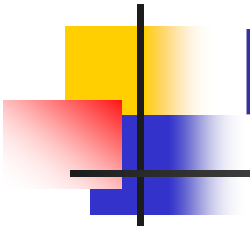
Object Adapter

- An object adapter bridges the gap between
  - CORBA objects with IDL interfaces and the programming language interfaces of the corresponding servant (classes).
  - It does the work of the remote reference and dispatcher modules in RMI.
- An object adapter has the following tasks:
  - It creates remote object references for CORBA objects;
  - It dispatches each RMI via a skeleton to the appropriate servant;
  - It activates objects.



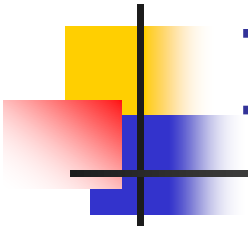
Object Adapter

- An object adapter gives each CORBA object a unique object name.
  - The same name is used each time an object is activated.
    - It is specified by the application program or generated by the object adapter.
  - Each active CORBA object is registered with its object adapter,
    - Which keeps a remote object table to map names of CORBA objects to servants.
- Each object adapter has its own name - specified by the application program or generated automatically.
- The object adapter has a remote object reference and a method to be invoked on it.



Portable Object Adapter

- Basic Object Adapter (BOA), a concrete specification
  - The first defined object adapter for use in CORBA-compliant ORBs
  - Leaves many features unsupported, requiring proprietary extensions
  - Superseded by the Portable Object Adapter (POA), facilitating server-side ORB-neutral code
- The ***Portable Object Adapter***, or ***POA***, is a particular type of object adapter that is defined by the CORBA specification. It allows an object implementation to function with different ORBs, hence the word portable.



Implementation Repository

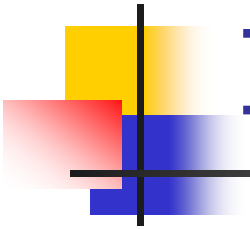
- Implementation repository
  - Activates registered servers on demand and locates running servers
  - Uses the object adapter name to register and activate servers.
  - Stores a mapping from the names of object adapters to the pathnames of files containing object implementations.
    - When a server program is installed it can be registered with the implementation repository.
    - When an object implementation is activated in a server, the hostname and port number of the server are added.

Implementation Repository

- Implementation repository entry:

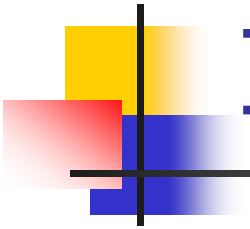
object adapter name	pathname of object implementation	hostname and port number of server

- Not all CORBA objects (e.g. call backs) need be activated on demand
- Access control information can be stored in an implementation repository



Interface Repository

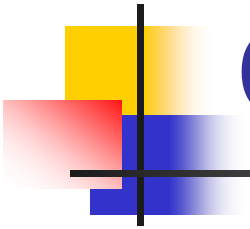
- Provides information about registered IDL interfaces
  - For an interface of a given type it can supply the names of the methods and for each method, the names and types of the arguments and exceptions.
  - A facility for reflection in CORBA.
  - If a client has a remote reference to a CORBA object, it can ask the interface repository about its methods and their parameter types
  - The client can use the dynamic invocation interface to construct an invocation with suitable arguments and send it to the server.



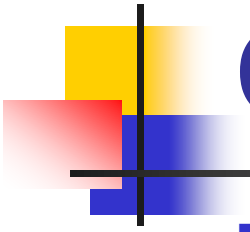
Interface Repository

- The IDL compiler gives a type identifier to each IDL type
- A type identifier is included in remote object references
- This type identifier is called the repository ID
  - Because the interface repository stoes interfaces against their IDs
- Applications that use static invocation with client proxies and IDL skeletons do not require an interface repository.
  - Not all ORBs provide an interface repository.

CORBA Naming Service



- CORBA specifies a generic directory service. The ***Naming Service*** serves as a directory for CORBA objects, and, as such, is platform independent and programming language independent.
- The Naming Service permits ORB-based clients to obtain references to objects they wish to use. It allows names to be associated with object references. Clients may query a naming service using a predetermined name to obtain the associated object reference.

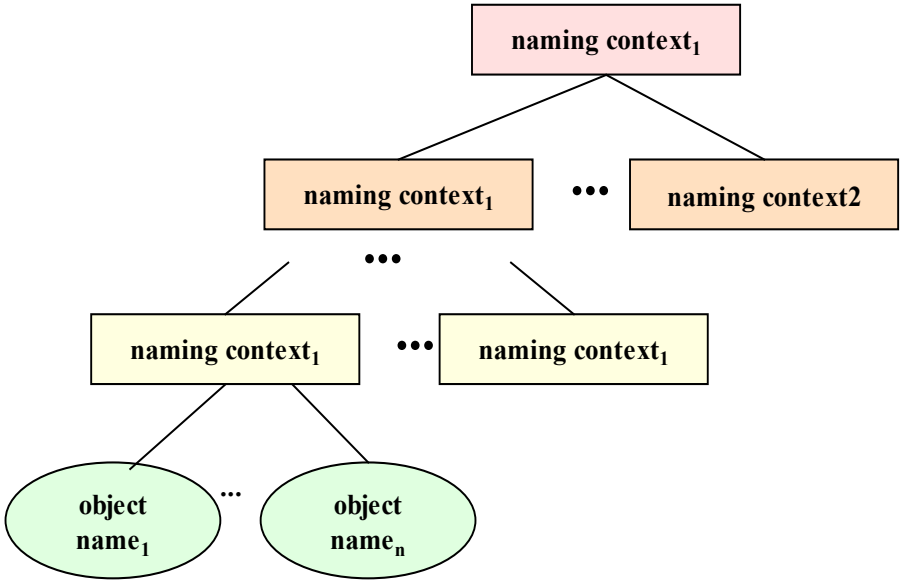


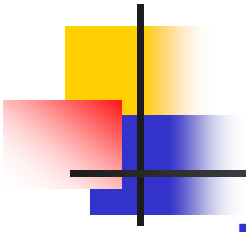
CORBA Naming Service

- To export a distributed object, a CORBA object server contacts a Naming Service to ***bind*** a symbolic name to the object. The Naming Service maintains a database of names and the objects associated with them.
- To obtain a reference to the object, an object client requests the Naming Service to look up the object associated with the name (This is known as ***resolving*** the object name.)
- The API for the Naming Service is specified in interfaces defined in IDL, and includes methods that allow servers to bind names to objects and clients to resolve those names.

CORBA Naming Service

To be as general as possible, the CORBA object naming scheme is necessarily complex. Since the name space is universal, a standard naming hierarchy is defined in a manner similar to the naming hierarchy in a file directory

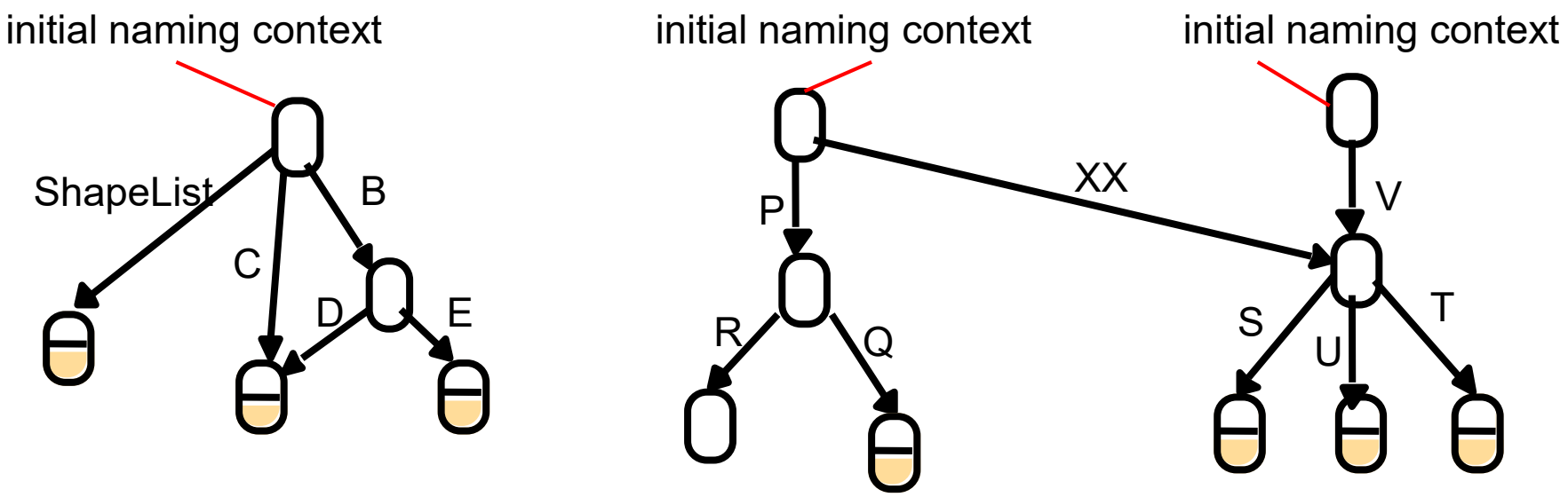




A Naming Context

- A naming context correspond to a folder or directory in a file hierarchy, while object names corresponds to a file.
- The full name of an object, including all the associated naming contexts, is known as a *compound name*. The first component of a compound name gives the name of a naming context, in which the second component is accessed. This process continues until the last component of the compound name has been reached.
- Naming contexts and name bindings are created using methods provided in the Naming Service interface.

Naming graph in CORBA Naming Service



CORBA Naming Service NamingContext interface in IDL

struct NameComponent { string id; string kind; };
typedef sequence <NameComponent> Name;
interface NamingContext {

void bind (in Name n, in Object obj);

binds the given name and remote object reference in my context.

void unbind (in Name n);

removes an existing binding with the given name.

void bind_new_context(in Name n);

creates a new naming context and binds it to a given name in my context.

Object resolve (in Name n);

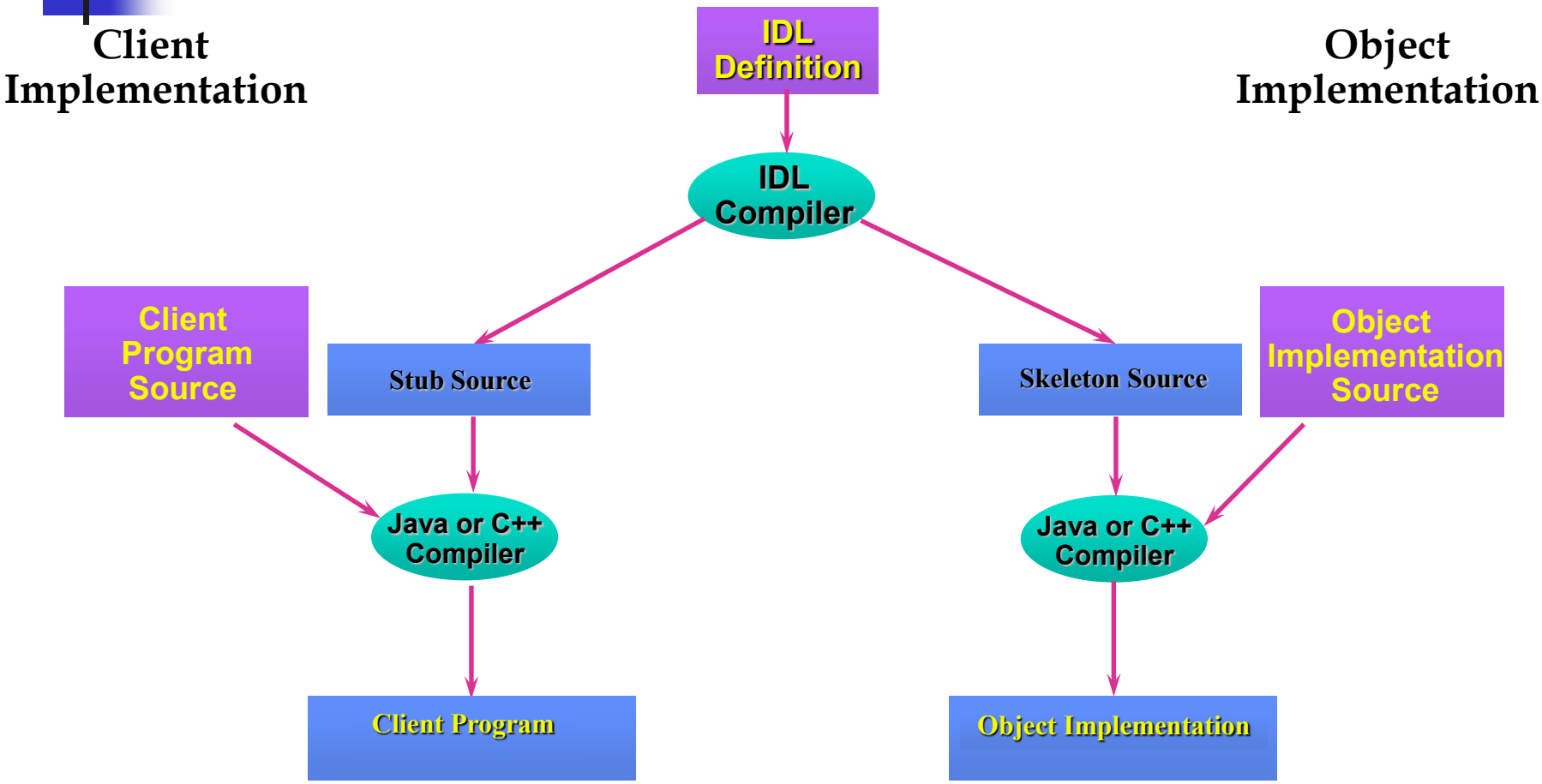
looks up the name in my context and returns its remote object reference.

void list (in unsigned long how_many, out BindingList bl, out BindingIterator bi);

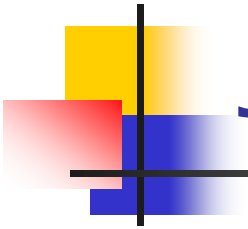
returns the names in the bindings in my context.

};

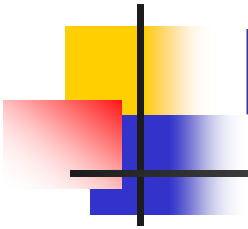
CORBA Development Process Using IDL



Java IDL – Java's CORBA Facility

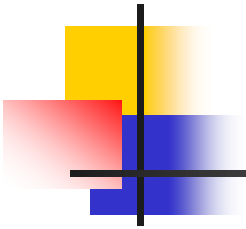


- IDL is part of the Java 2 Platform, Standard Edition (J2SE).
- The Java IDL facility includes a CORBA Object Request Broker (ORB), an IDL-to-Java compiler, and a subset of CORBA standard services.
- In addition to the Java IDL, Java provides a number of CORBA-compliant facilities, including ***RMI over IIOP***, which allows a CORBA application to be written using the RMI syntax and semantics.



Key Java IDL Packages

- package org.omg.CORBA – contains interfaces and classes which provides the mapping of the OMG CORBA APIs to the Java programming language
- package org.omg.CosNaming - contains interfaces and classes which provides the naming service for Java IDL
- org.omg.CORBA.ORB - contains interfaces and classes which provides APIs for the Object Request Broker.



Java IDL Tools

Java IDL provides a set of tools needed for developing a CORBA application:

- idlj - the IDL-to-Java compiler (called idl2java in Java 1.2 and before)
- orbd - a server process which provides Naming Service and other services
- servertool – provides a command-line interface for application programmers to register/unregister an object, and startup/shutdown a server.
- tnameserv – an olderTransient Java IDL Naming Service whose use is now discouraged.

The CORBA Interface file Hello.idl

```
module HelloApp
{
interface Hello
   {
      string sayHello();
      oneway void shutdown();
   };
};
```

Compiling the IDL file

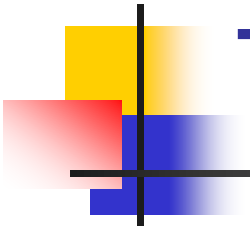
- The IDL file should be placed in a directory dedicated to the application. The file is compiled using the compiler ***idlj*** using a command as follows:

  idlj -fall Hello.idl

  The –***fall*** command option is necessary for the compiler to generate all the files needed.
- In general, the files can be found in a subdirectory named <some name>App when an interface file named <some name>.idl is compiled.
- If the compilation is successful, the following files can be found in a ***HelloApp*** subdirectory:

  HelloOperations.java ***Hello.java***
  HelloHelper.java ***HelloHolder.java***
  _HelloStub.java ***HelloPOA.java***

  These files require no modifications.



The *Operations.java file

- There is a file HelloOperations.java found in HelloApp/ after you compiled using idlj
- It is known as a ***Java operations interface*** in general
- It is a Java interface file that is equivalent to the CORBA IDL interface file (***Hello.idl***)
- You should look at this file to make sure that the method signatures correspond to what you expect.

HelloApp/HelloOperations.java

The file contains the methods specified in the original IDL file: in this case the methods ***sayHello()*** and ***shutdown()***.

```
package HelloApp;
  /**
  * HelloApp/HelloOperations.java Generated by the
  * IDL-to-Java compiler (portable), version "3.1" from Hello.idl
  */

  public interface HelloOperations
  {
      String sayHello ();
      void shutdown ();
  } // interface HelloOperations
```

HelloApp/Hello.java

The signature interface file combines the characteristics of the Java *operations* interface (*HelloOperations.java*) with the characteristics of the CORBA classes that it extends.

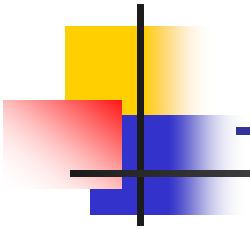
```
package HelloApp;
/**
* HelloApp/Hello.java
* Generated by the IDL-to-Java compiler (portable),
* version "3.1" from Hello.idl
*/
public interface Hello extends HelloOperations,
    org.omg.CORBA.Object,
    org.omg.CORBA.portable.IDLEntity
{ …
} // interface Hello
```

HelloHelper.java, the Helper class

- The Java class `HelloHelper` provides auxiliary functionality needed to support a CORBA object in the context of the Java language.
- In particular, a method, ***`narrow`***`,` `allows a` CORBA object reference to be cast to its corresponding type in Java, so that a CORBA object may be operated on using the syntax for Java object.

HelloHolder.java, the Holder class

- The Java class called `HelloHolder` holds (contains) a reference to an object that implements the `Hello interface`.
- The class is used to handle an `out` or an `inout` parameter in IDL in Java syntax (In IDL, a parameter may be declared to be ***out*** if it is an output argument, and ***inout*** if the parameter contains an input value as well as carries an output value.)

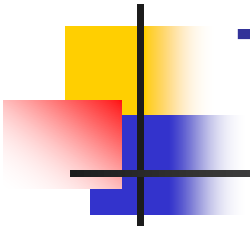


_HelloStub.java

- The Java class ***HelloStub*** is the stub file, the client-side proxy, which interfaces with the client object.
- It extends `org.omg.CORBA.portable.ObjectImpl` and implements the ***Hello.java*** interface.

HelloPOA.java, the server skeleton

- The Java class ***HelloImplPOA*** is the skeleton, the server-side proxy, combined with the portable object adapter.
- It extends ***org.omg.PortableServer.Servant***, and implements the ***InvokeHandler*** interface and the ***HelloOperations*** interface.



The application

Server-side Classes

- On the server side, two classes need to be provided: the servant and the server.
- The servant, *HelloImpl*, is the implementation of the *Hello* IDL interface; each *Hello* object is an instantiation of this class.

The Servant - HelloApp/HelloImpl.java

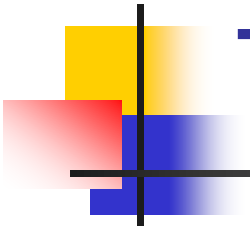
```
// The servant -- object implementation -- for the Hello
// example.  Note that this is a subclass of HelloPOA,
// whose source file is generated from the
// compilation of Hello.idl using j2idl.
     import HelloApp.*;
     import org.omg.CosNaming.*;
     import java.util.Properties; …
     class HelloImpl extends HelloPOA {
     private ORB orb;
       public void setORB(ORB orb_val) {
         orb = orb_val;
       }
      // implement sayHello() method
      public String sayHello() {
         return "\nHello world !!\n";
      }
      // implement shutdown() method
      public void shutdown() {
        orb.shutdown(false);
      }
    } //end class
```

The server - HelloApp/HelloServer.java

```
public class HelloServer {
  public static void main(String args[]) {
   try{
   // create and initialize the ORB
   ORB orb = ORB.init(args, null);
   // get reference to rootpoa & activate the POAManager
   POA rootpoa =
   (POA)orb.resolve_initial_references("RootPOA");
   rootpoa.the_POAManager().activate();
   // create servant and register it with the ORB
   HelloImpl helloImpl = new HelloImpl();
   helloImpl.setORB(orb);
   // get object reference from the servant
   org.omg.CORBA.Object ref =
      rootpoa.servant_to_reference(helloImpl);
   // and cast the reference to a CORBA reference
   Hello href = HelloHelper.narrow(ref);
```

HelloApp/HelloServer.java - continued

```
// get the root naming context
// NameService invokes the transient name service
org.omg.CORBA.Object objRef =
  orb.resolve_initial_references("NameService");
// Use NamingContextExt, which is part of the
// Interoperable Naming Service (INS) specification.
NamingContextExt ncRef =
NamingContextExtHelper.narrow(objRef);
// bind the Object Reference in Naming
String name = "Hello";
NameComponent path[] = ncRef.to_name( name );
ncRef.rebind(path, href);
System.out.println("HelloServer ready and waiting ...");
// wait for invocations from clients
  orb.run();
```



The object client application

- A client program can be a Java application, an applet, or a servlet.
- The client code is responsible for creating and initializing the ORB, looking up the object using the Interoperable Naming Service, invoking the narrow method of the ***Helper*** object to cast the object reference to a reference to a ***Hello*** object implementation, and invoking remote methods using the reference. The object's ***sayHello*** method is invoked to receive a string, and the object's shutdown method is invoked to deactivate the service.

The object client application

```
// A sample object client application.
import HelloApp.*;
import org.omg.CosNaming.*; …
public class HelloClient{
  static Hello helloImpl;
  public static void main(String args[]){
     try{
      ORB orb = ORB.init(args, null);
      org.omg.CORBA.Object objRef =
         orb.resolve_initial_references("NameService");
      NamingContextExt ncRef =
         NamingContextExtHelper.narrow(objRef);
      helloImpl =
         HelloHelper.narrow(ncRef.resolve_str("Hello"));
      System.out.println(helloImpl.sayHello());
      helloImpl.shutdown();
```

Compiling and Running a Java IDL application

1. Create and compile the Hello.idl file on the server machine:

   idlj -fall Hello.idl

2. Copy the directory containing Hello.idl (including the subdirectory generated by ***idlj***) to the client machine.
3. In the ***HelloApp*** directory on the client machine: create ***HelloClient.java***. Compile the *.java files, including the stubs and skeletons (which are in the directory ***HelloApp***):

   javac *.java HelloApp/*.java

Compiling and Running a Java IDL application

4. In the ***HelloApp*** directory on the server machine:
   - Create ***HelloServer.java***. Compile the `.java` files:

     javac *.java HelloApp/*.java
   - On the server machine: Start the Java Object Request Broker Daemon, ***orbd***, which includes a Naming Service.

     To do this in Unix:

     ```
     orbd -ORBInitialPort 1050
          -ORBInitialHost servermachinename&
     ```

     To do this in Windows:

     ```
     start orbd -ORBInitialPort 1050
          -ORBInitialHost servermachinename
     ```
5. On the server machine, start the Hello server, as follows:

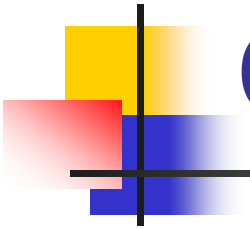
   ```
   java HelloServer –ORBInitialHost <nameserver
    host name> -ORBInitialPort 1050
   ```

Compiling and Running a Java IDL application

6. On the client machine, run the *Hello* application client. From a DOS prompt or shell, type:

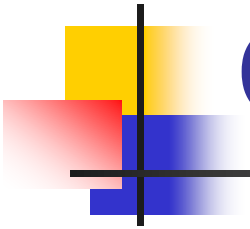
   ```
   java HelloClient -ORBInitialHost
         nameserverhost -ORBInitialPort 1050
   ```

   all on one line. Note that *nameserverhost* is the host on which the IDL name server is running. In this case, it is the server machine.
7. Kill or stop *orbd* when finished. The name server will continue to wait for invocations until it is explicitly stopped.
8. Stop the object server.



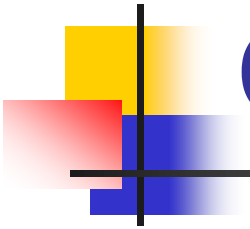
CORBA Object

- The term *CORBA object* is used to refer to remote objects.
  - A *CORBA object* implements an IDL interface, has a remote object reference and its methods can be invoked remotely.
- A CORBA object can be implemented by a language without classes.
  - The class concept does not exist in CORBA.
  - Therefore classes cannot be defined in CORBA IDL, which means that instances of classes cannot be passed as arguments.



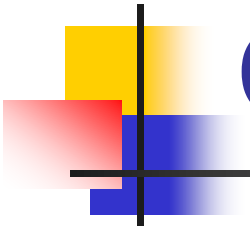
CORBA object characteristics

- CORBA objects have identity
  - A CORBA server can contain multiple instances of multiple interfaces
  - An IOR uniquely identifies one object instance
- CORBA object references can be persistent
  - Some CORBA objects are transient, short-lived and used by only one client
  - But CORBA objects can be shared and long-lived
    - Business rules and policies decide when to "destroy" an object
    - IORs can outlive client and even server process life spans



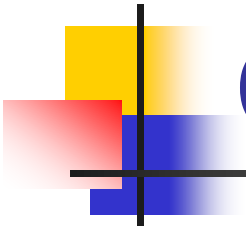
CORBA object characteristics

- CORBA objects can be relocated
  - The fixed object key of an object reference does not include the object's location
  - CORBA objects may be relocated at admin time or runtime
  - ORB implementations may support the relocation transparently
- CORBA supports replicated objects
  - IORs with the same object key but different locations are considered replicas



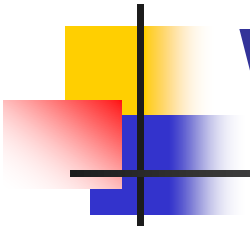
CORBA server characteristics

- When we say "server" we usually mean server process, not server machine
- One or more CORBA server processes may be running on a machine
- Each CORBA server process may contain one or more CORBA object instances, of one or more CORBA interfaces
- A CORBA server process does not have to be "heavyweight"
  - e.g., a Java applet can be a CORBA server



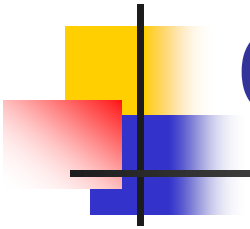
CORBA Object Interface

- A distributed object is defined using a software file similar to the remote interface file in Java RMI.
- Since CORBA is language independent, the interface is defined using a universal language with a distinct syntax, known as the ***CORBA Interface Definition Language*** (***IDL***).
- The syntax of CORBA IDL is similar to Java and C++. However, object defined in a CORBA IDL file can be implemented in a large number of diverse programming languages, including C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python, and IDLScript.



Why IDL?

- IDL reconciles diverse object models and programming languages
- Imposes the same object model on all supported languages
- Programming language independent means of describing data types and object interfaces
  - Purely descriptive - no procedural components
  - Provides abstraction from implementation
  - Allows multiple language bindings to be defined
- A means for integrating and sharing objects from different object models and languages



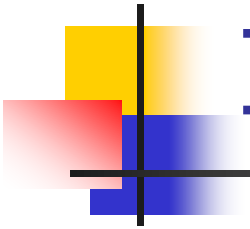
CORBA IDL

- IDL provides facilities for defining modules, interfaces, types, attributes and method signatures.
- IDL has the same lexical rules as C++ but has additional keywords to support distribution,
  - for example *interface*, *any*, *attribute*, *in*, *out*, *inout*, *readonly*, *raises*.
- It allows standard C++ pre-processing facilities.
- The grammar of IDL is a subset of ANSI C++ with additional constructs to support method signatures.

CORBA Methods and Modules

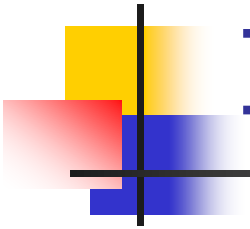
[oneway] <return_type> <method_name> (parameter1,..., parameterL)
[raises (except1,..., exceptN)]
[context (name1,..., nameM)]

- Each parameter is labelled as *in*, *out* or *inout*
  - *e.g. void getPerson(in string name, out Person p);*
- *oneway* method
  - The client will not be blocked and *maybe* semantics is used
  - At-most-once call semantics is the default
- Modules allow interfaces and associated definitions to be grouped.



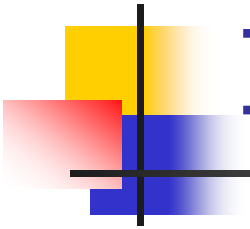
IDL simple data types

- Basic data types similar to C, C++ or Java
  - long, long long, unsigned long, unsigned long long
  - short, unsigned short
  - float, double, long double
  - char, wchar (ISO Unicode)
  - boolean
  - octet (raw data without conversion)
  - any (self-describing variable)



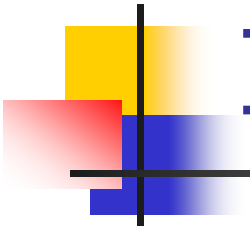
IDL complex data types

- string - sequence of characters - bounded or unbounded
  - string<256> msg // bounded
  - string msg // unbounded
- wstring - sequence of Unicode characters - bounded or unbounded
- sequence - one dimensional array whose members are all of the same type - bounded or unbounded
  - sequence<float, 100> mySeq // bounded
  - sequence<float> mySeq // unbounded



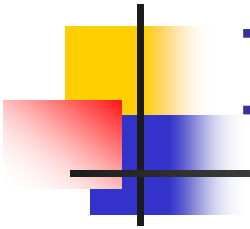
IDL user defined data types

- Facilities for creating your own types:
  - typedef
  - enum
  - const
  - struct
  - union
  - arrays
  - exception
- Preprocessor directives - #include #define



Inheritance

- IDL interfaces may extend one or more interfaces
  - All IDL interfaces are compatible with *Object*
    - Can use type *Object* for parameters that may be of any type e.g. *bind* and *resolve* in the Naming Service
  - An extended interface may add new methods, types, constants and exceptions
  - It may redefine types, constants and exceptions but not methods



IDL constructed types – 1

Type	*Examples*	*Use*
sequence	*typedef sequence <Shape, 100> All;* *typedef sequence <Shape> All* bounded and unbounded sequences of *Shapes*	Defines a type for a variable-length sequence of elements of a specified IDL type. An upper bound on the length may be specified.
string	*String name;* *typedef string<8> SmallString;* unbounded and bounded sequences of characters	Defines a sequences of characters, terminated by the null character. An upper bound on the length may be specified.
array	*typedef octet uniqueId[12];* *typedef GraphicalObject GO[10][8]*	Defines a type for a multi-dimensional fixed-length sequence of elements of a specified IDL type.

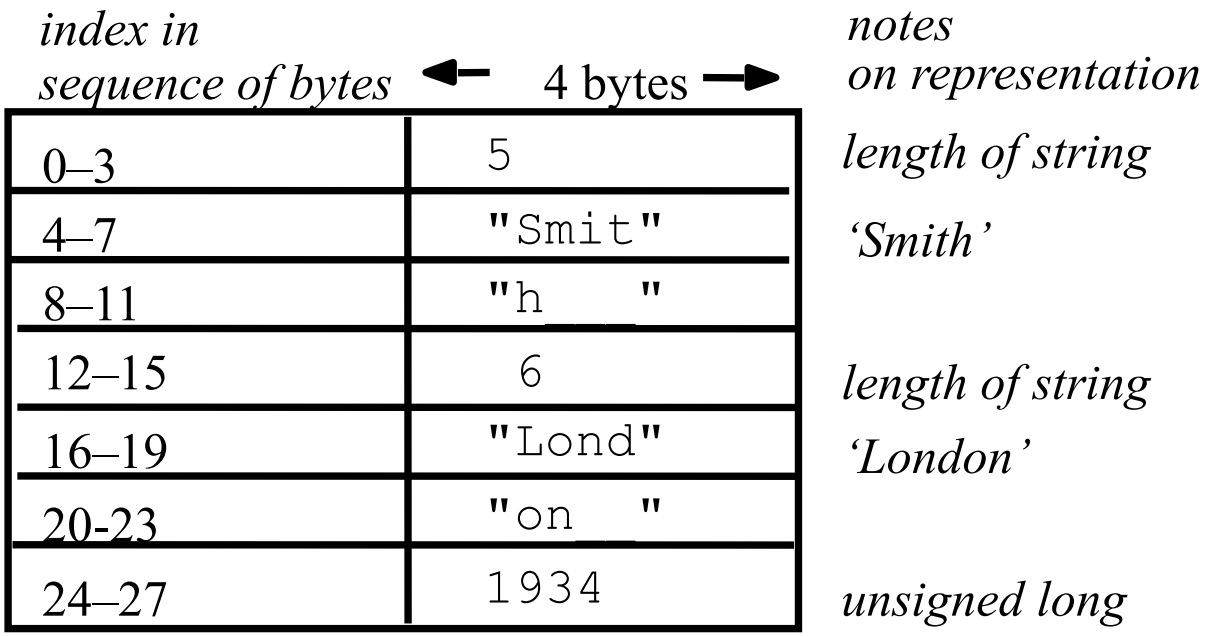
IDL constructed types – 2

Type	*Examples*	*Use*
record	*struct GraphicalObject {* *string type;* *Rectangle enclosing;* *boolean isFilled;* *};*	Defines a type for a record containing a group of related entities. *Structs* are passed by value in arguments and results.
enumerated	*enum Rand* *(Exp, Number, Name);*	The enumerated type in IDL maps a type name onto a small set of integer values.
union	*union Exp switch (Rand) {* *case Exp: string vote;* *case Number: long n;* *case Name: string s;* *};*	The IDL discriminated union allows one of a given set of types to be passed as an argument. The header is parameterized by an *enum*, which specifies which member is in use.

CORBA CDR for constructed types

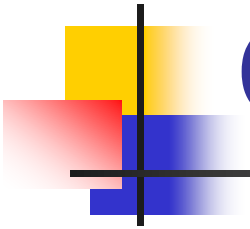
Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

CORBA CDR message



index in sequence of bytes	← 4 bytes →	*notes on representation*
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20-23	"on__"	
24–27	1934	*unsigned long*

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1934}

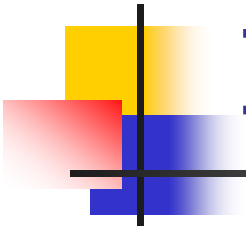


Operations and parameters

- Return type of operations can be any IDL type
- Each parameter has a direction (in, out, inout) and a name
- Similar to C/C++ function declarations

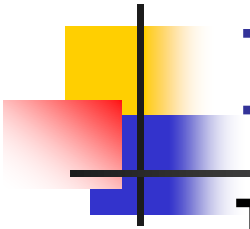
Parameters in CORBA IDL

- Passing CORBA objects:
  - Any parameter or return value whose type is specified by the name of a IDL interface, e.g. *Shape*, is a reference to a CORBA object (see *newShape*)
  - And the value of a remote object reference is passed.
- Passing CORBA primitive and constructed types:
  - Arguments of primitive and constructed types are copied and passed by value. On arrival, a new value is created in the recipient's process. E.g., the *struct GraphicalObject* (argument of *newShape* and result of *getAllState*)
- Type *Object* - is a supertype of all IDL interfaces (its values are object references).



Inter-ORB Protocols

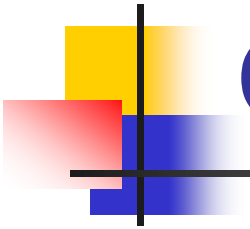
- To allow ORBs to be interoperable, the OMG specified a protocol known as the ***General Inter-ORB Protocol*** (***GIOP***), a specification which "provides a general framework for protocols to be built on top of specific transport layers."
- A special case of the protocol is the ***Internet Inter-ORB Protocol*** (***IIOP***), which is the GIOP applied to the TCP/IP transport layer.



Inter-ORB Protocols

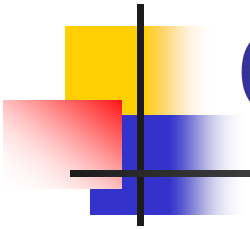
The IIOP specification includes :

1. **Transport management requirements**: specifies the connection and disconnection requirements, and the roles for the object client and object server in making and unmaking/braking connections.
2. **Definition of common data representation**: a coding scheme for marshalling and unmarshalling data of each IDL data type.
3. **Message formats**: different types of message format are defined. The messages allow clients to send requests to object servers and receive replies. A client uses a Request message to invoke a method declared in a CORBA interface for an object and receives a reply message from the server.



Object Servers and Object Clients

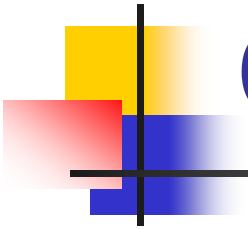
- As in Java RMI, a CORBA distributed object is exported by an ***object server***, similar to the object server in RMI.
- An ***object client*** retrieves a reference to a distributed object from a naming or directory service, to be described, and invokes the methods of the distributed object.



CORBA Object References

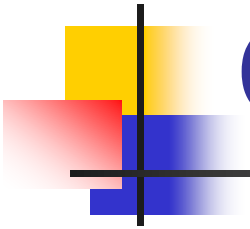
- As in Java RMI, a CORBA distributed object is located using an ***object reference***. Since CORBA is language-independent, a CORBA object reference is an abstract entity mapped to a language-specific object reference by an ORB, in a representation chosen by the developer of the ORB.
- For interoperability, OMG specifies a protocol for the abstract CORBA object reference object, known as the ***Interoperable Object Reference*** (***IOR***) protocol.

Interoperable Object Reference (IOR)



- For interoperability, OMG specifies a protocol for the abstract CORBA object reference object, known as the ***Interoperable Object Reference*** (***IOR***) protocol.
- An ORB compatible with the IOR protocol will allow an object reference to be registered with and retrieved from any IOR-compliant directory service. CORBA object references represented in this protocol are called ***Interoperable Object Reference***s (***IOR***s).

Interoperable Object Reference (IOR)



An IOR is a string that contains encoding for the following information:

- The type of the object.
- The host where the object can be found.
- The port number of the server for that object.
- An object key, a string of bytes identifying the object. The object key is used by an object server to locate the object.

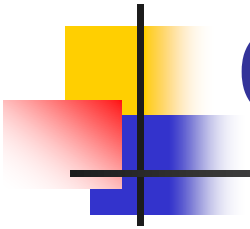
Interoperable Object Reference (IOR)

IOR format

IDL interface type name	Protocol and address details			Object key	
interface repository identifier	IIOP	host domain name	port number	adapter name	object name

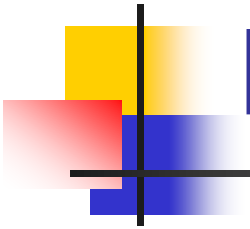
The following is an example of the string representation of an IOR:

```
IOR:000000000000000d49444c3a677269643a312e3000000
00000000010000000000000004c000100000000000015756c74
72612e6475626c696e2e696f6e612e6965000009630000002
83a5c756c7472612e6475626c696e2e696f6e612e69653a67
7269643a303a3a49523a6772696400 3a
```



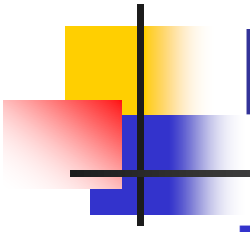
CORBA services

- The OMG has defined a set of Common Object Services
- Frequently used components needed for building robust applications
- Typically supplied by vendors
- OMG defines interfaces to services to ensure interoperability



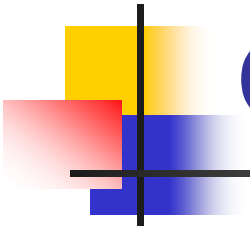
Popular CORBA services

- Naming
  - maps logical names to to server objects
  - references may be hierarchical, chained
  - returns object reference to requesting client
- Events
  - asynchronous messaging
  - decouples suppliers and consumers of information



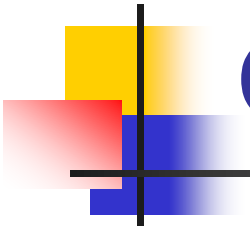
Popular CORBA services

- Notification
  - More robust enhancement of event service
  - Quality of Service properties
  - Event filtering
  - Structured events
- Transaction
  - Ensures correct state of transactional objects
    - Manages distributed commit/rollback
    - Implements the protocols required to guarantee the ACID (Atomicity, Consistency, Isolation, and Durability) properties of transactions



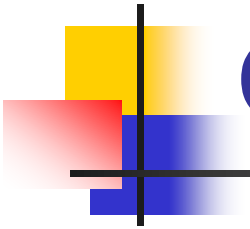
CORBA vs. ad-hoc networked apps

- Technical considerations:
  - CORBA/EJB implementations have integration with object databases, transaction services, security services, directory services, etc.
  - CORBA implementations automatically optimize transport and marshalling strategies
  - CORBA implementations automatically provide threading models



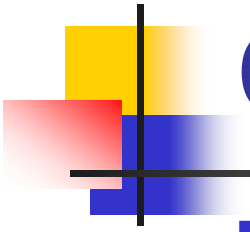
CORBA vs. ad-hoc networked apps

- Business considerations:
  - Standards based
  - Multiple competing interoperable implementations
  - Buy vs. build tradeoffs
  - Resource availability
    - Software engineers
    - Tools



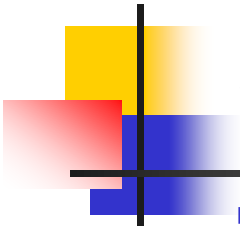
CORBA vs. Java RMI

- CORBA differs from the architecture of Java RMI in one significant aspect:
  - RMI is a proprietary facility developed by Sun MicroSystems, Inc., and supports objects written in the Java programming langugage only.
  - CORBA is an architecture that was developed by the Object Management Group (OMG), an industrial consortium.



CORBA, J2EE and .NET

- CORBA
  - MANY LANGUAGES, MANY PLATFORMS
    - Vendor-neutral specification
    - Dominates infrastructural "backbone" of distributed services
- Java technologies (EJB, J2EE, …)
  - ONE LANGUAGE, MANY PLATFORMS
    - Utilise "write once, read everywhere" philosophy of Java
    - Must be implemented in Java
    - EJB standard a subset of CCM since April, 1999
- COM+ (.NET)
  - MANY LANGUAGES, ONE PLATFORM
    - Binary specification, language-neutral
    - Dominates desktop
    - Centre of gravity is Windows environment

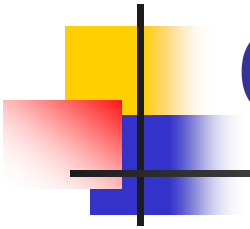


CORBA, J2EE, and .NET

- CORBA, J2EE are NOT products
  - They are open specifications
- .NET is a PRODUCT STRATEGY
  - Specification is partly open, partly proprietary
- CORBA, J2EE are multi-vendor, mature standards supported by 30+ vendors
  - Sophisticated programming model
  - Better leverage of legacy
- .NET is a single vendor, integrated, dedicated web services strategy
  - Simpler programming model
  - Leverages knowledge of Win32
- J2EE, CORBA are generalised enterprise-wide software strategies
  - Web services are a late "bolt-on" systems
- .NET is multi-language, J2EE is single language
  - .NET has a better "shared business context" story
  - J2EE has a better "shared technical context" story
- .NET is being marketed by Microsoft's A team
- J2EE (and CORBA) is being marketed by an entire industry

Client Callback in CORBA

```
// Illustrates CORBA callback -- sample obtained via
// http://java.sun.com/products/jdk/1.2/docs/guide/idl/jidlExample3.html

module HelloApp
{
    interface HelloCallback
    {
        void callback(in string message);
    };


    interface Hello
    {
        string sayHello(in HelloCallback objRef, in string message);
    };
};
```



Callback Hello Client

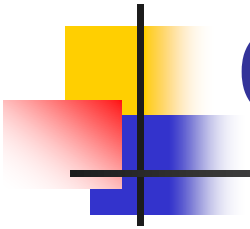
```
// Illustrates CORBA callback -- sample obtained via
// http://java.sun.com/products/jdk/1.2/docs/guide/idl/jidlExample3.html

import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

class HelloCallbackServant extends _HelloCallbackImplBase
{
    public void callback(String notification)
    {
        System.out.println(notification);
    }
}
```

Callback Hello Client, Continued

```
public class HelloClient
{
    public static void main(String args[])
    {
     try{
            // create and initialize the ORB
            ORB orb = ORB.init(args, null);
            // get the root naming context
            org.omg.CORBA.Object objRef =
                                orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);

            // resolve the Object Reference in Naming
            NameComponent nc = new NameComponent("Hello", "");
            NameComponent path[] = {nc};
            Hello helloRef = HelloHelper.narrow(ncRef.resolve(path));
```

Callback Hello Client, Continued

```
HelloCallbackServant helloCallbackRef = new HelloCallbackServant();
orb.connect(helloCallbackRef);

        // call the Hello server object and print results
        String hello = helloRef.sayHello(helloCallbackRef,"\ntest..\n");
        System.out.println(hello);

    } catch (Exception e) {
        System.out.println("ERROR : " + e) ;
        e.printStackTrace(System.out);
    }
   }
}
```



Callback Hello Server

```
// Illustrates CORBA callback -- sample obtained via
// http://java.sun.com/products/jdk/1.2/docs/guide/idl/jidlExample3.html

import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

class HelloServant extends _HelloImplBase
{
    public String sayHello(HelloCallback callobj, String msg)
    {
     callobj.callback(msg);
     return "\nHello world !!\n";
    }
}
```

Callback Hello Server, Continued

```
public class HelloServer {

    public static void main(String args[])
    {
     try{
        // create and initialize the ORB
        ORB orb = ORB.init(args, null);

        // create servant and register it with the ORB
        HelloServant helloRef = new HelloServant();
        orb.connect(helloRef);

        // get the root naming context
        org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
        NamingContext ncRef = NamingContextHelper.narrow(objRef);
```

Callback Hello Server, Continued

```
// bind the Object Reference in Naming
        NameComponent nc = new NameComponent("Hello", "");
        NameComponent path[] = {nc};
        ncRef.rebind(path, helloRef);

        // wait for invocations from clients
           java.lang.Object sync = new java.lang.Object();
           synchronized (sync) {
               sync.wait();
           }
     } catch (Exception e) {
        System.err.println("ERROR: " + e);
        e.printStackTrace(System.out);
     }
    }
}
```